Scheduling Recurrent Precedence-Constrained Task Graphs on a Symmetric Shared-Memory Multiprocessor

UmaMaheswari C. Devi

IBM India Research Laboratory, Bangalore, India

Abstract. We consider approaches that allow task migration for scheduling recurrent directed-acyclic-graph (DAG) tasks on symmetric, shared-memory multiprocessors (SMPs) in order to meet a given throughput requirement with fewer processors. Within the scheduling approach proposed, we present a heuristic based on grouping DAG subtasks for lowering the end-to-end latency and an algorithm for computing an upper bound on latency. Unlike prior work, the purpose of the grouping here is not to map the subtask groups to physical processors, but to generate aggregated entities, each of which can be treated as a single schedulable unit to lower latency. Evaluation using synthetic task sets shows that our approach can lower processor needs considerably while incurring only a modest increase in latency. In contrast to the work presented herein, most prior work on scheduling recurrent DAGs has been for distributed-memory multiprocessors, and has therefore mostly been concerned with statically mapping DAG subtasks to processors.

1 Introduction

Symmetric, shared-memory multiprocessor (SMP) systems, including those based on multicore processors, are now mainstream. With Moore's law manifesting in the form of increasing number of processing cores per chip (as opposed to faster individual cores), only applications with parallelism and to which the available processing resources can be allocated intelligently are likely to witness processing speed-ups. Some applications that can benefit from the current and emerging SMPs include image and digital signal processing systems, deep packet inspection in computer networks, etc.

Several applications like those specified above can be modeled as *directed-acyclic-graphs* (DAGs), also referred to as *task graphs*. In a task graph, nodes denote *subtasks*, and edges, the order in which subtasks should execute on a given input. A simple DAG with six nodes, whose weights denote their execution times, is shown in Fig. 1. Path branches and merges in the DAG are assumed to have fork and join semantics; branches therefore provide *spatial concurrency*. External inputs to some of these systems are recurrent, arriving periodically, such as sampled radar pulses fed into an embedded signal processing system at a specified rate. The external input rate dictates the throughput that should be met. In some systems, processing of later inputs may overlap that of earlier ones, allowing subtasks to be pipelined and enabling *temporal concurrency*. A deadline may also be imposed on the end-to-end latency incurred in processing any given input. Scheduling such recurrent DAGs with spatial and temporal parallelism on SMPs is the subject of this paper.

H. Sips, D. Epema, and H.-X. Lin (Eds.): Euro-Par 2009, LNCS 5704, pp. 265–280, 2009.
© Springer-Verlag Berlin Heidelberg 2009

The problem of scheduling DAGs on multiprocessors has received considerable attention. Much of the early work was limited to scheduling a single DAG instance, while some later and recent work deals with recurring instances. This later work is however focused on distributed-memory multiprocessor (DMM) platforms, on which inter-processor communication and migration costs can be significant. Hence, most of prior work statically maps subtasks onto processors and executes each subtask on its assigned processors only.

Inter-processor communication and migration costs are, in general, less of an issue on SMPs with uniform memory access times, and, in particular, almost negligible on multicore processors with shared caches [6]. Therefore, techniques proposed in the context of DMMs may be overkill for SMPs and need reevaluation for SMPs. An example of the latter class of processors is Sun's eight-core UltraSPARC T1 Niagara chip, whose cores all share a common L2 cache. Also, the number of cores per chip is expected to increase with passing years. In light of these developments, we consider a more flexible scheduling approach in which a given instance or different instances of a subtask may execute on different processors, as opposed to statically binding a subtask to a processor.

Contributions. First, we draw upon optimal multiprocessor rate-based scheduling algorithms from the real-time scheduling literature and let subtasks of a DAG to migrate to minimize the processor needs on an SMP while meeting a stipulated throughput. E.g., in Fig. 1, if external inputs arrive at T once every three time units, then static mapping of subtasks will require at least *six* processors to keep up with the incoming requests. On the other hand, if subtasks can migrate, then they may be scheduled on only *four* processors such that each subtask executes for at least two time units in intervals spanning three time units, which is sufficient to guarantee stability. In this example, processor needs are higher by 50% if migrations are disallowed. On the other hand, lowering the number of processors has the effect of worsening T's end-to-end latency from six to nine time units. Hence, the approach may not be applicable to all systems, but, resource savings can be quite considerable if some increase in latency can be tolerated. E.g., in systems that consist of multiple tasks as T, each task could contribute to saving two processors, leading to substantial cumulative savings.

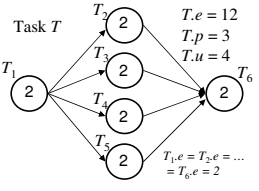

Fig. 1. Sample task graph T

Next, within the scheduling approach proposed, we consider lowering end-to-end latency and present a heuristic for the purpose. Our heuristic is based on grouping consecutive subtasks on a sub-path into a task chain that may be scheduled as a single unit. Unlike prior work, the purpose of our grouping is not to determine a mapping from subtasks to processors. Thirdly, we provide an algorithm for computing an upper bound on the end-to-end latency for any input through the DAG under the scheduling approach and heuristic proposed. Finally, we evaluate the efficacy of our methods using synthetic task graphs.

Organization. The rest of this paper is organized as follows. Our task and system model are presented in Sec. 2, followed by a description of the basic scheduling

approach that minimizes processor needs in Sec. 3. Heuristic for lowering latency and an algorithm to compute an upper bound on latency are presented in Sec. 4. Sec. 5 provides a simulation-based evaluation, Sec. 6 reviews related work, while Sec. 7 concludes.

2 Task and System Model

We consider scheduling *tasks* modeled as DAGs, also referred to as *task graphs*, on SMPs. Each node of a DAG denotes a *subtask*, which constitutes a sequential section of an entire task. Tasks are denoted using upper-case letters; subtasks are indexed and denoted, *e.g.*, as $T_1, T_2, \ldots$. Since a subtask is sequential, it may not execute on more than one processor at a time although execution on different processors at different times is allowed. Nodes are weighted, with the weight of a node T_i denoting the associated subtask's *worst-case execution cost* $T_i.e$.

A DAG's edges impose a partial order among its nodes, indicating the orders in which subtasks may execute while processing a single request. Branches in paths have fork semantics, hence, all branches should be executed, and may proceed concurrently, if enough resources are available. Inter-processor communication costs are assumed to be negligible, so edges are not weighted. An example task graph with six nodes T_1–T_6 is shown in Fig. 1.

A task is recurrent and triggered by external inputs or *arrivals*. We assume that any two consecutive arrivals to a task T are separated by at least its minimum inter-arrival time, referred to as its *period*, denoted $T.p$. This can be accomplished in practical implementations by delaying an early arrival until its stipulated arrival time (assuming that the arrival rate over the long run does not exceed $1/period$). A task's period may be smaller than the weighted length of any of the paths in its graph, necessitating that subtasks be pipelined. However, a single subtask cannot have more than one instance executing at any time, which we refer to as *subtask concurrency constraint* (SCC). Therefore, a task's period $T.p$ may not be smaller than the execution costs of any of its subtasks T_i, $T_i.e$. The j^{th} instance of task T and subtask T_i processing the j^{th} arrival are denoted T^j and T_i^j, respectively.

The sum of the execution costs of all the subtasks of T is referred to as the *total execution cost* of T, denoted $T.e$. The ratio of $T.e$ to $T.p$ is the *utilization*, $T.u$, of T. Task preemption and migration costs are assumed to be negligible.

3 Basic Task Graph Scheduling Approach for SMPs

One trivial way of meeting a task T's throughput requirement, while respecting the SCC, is to order its subtasks in a linear chain that is consistent with the DAG's partial order, round robin external arrivals among available processors, and process each arrival exclusively on a single processor. E.g., in Fig. 1, $T.p = 3$, $T.u = 4$, so T can be scheduled on four processors by running instance $T^{4 \cdot i + j}$ on processor j for all $i \geq 0$ and $j = 1, 2, 3, 4$. That is, subtasks T_1^1–T_1^6 will be run in order on the first processor, T_2^1–T_2^6 on the second processor, and so on. It is easy to show that, by this method, it suffices to allocate $\lceil T.u \rceil$ processors to T.

The above method is optimal for task graphs that are linear chains with integral utilization, but not for arbitrary DAGs or chains with non-integral utilization. Also, end-to-end latency could be extremely high for arbitrary DAGs. E.g., latency for T in Fig. 1 is twelve by this approach, and would increase with additional nodes added to the concurrent stage, whereas the length of the longest path would remain at six. In this sense, the approach *does not scale* with increasing concurrency.

In what follows, we propose a scheduling approach that scales with concurrency. For this, we draw upon optimal multiprocessor rate-based scheduling algorithms. We begin with a brief overview.

3.1 Overview of Rate-Based Scheduling on Multiprocessors

Algorithms based on *proportionate-fair* or *Pfair scheduling* [8] are the only known way of *optimally* scheduling systems of stand-alone *rate-based tasks*, which can be thought of as a generalization of *sporadic tasks*, on multiprocessors. A sporadic task T is *sequential*, characterized by a period $T.p$ and a worst-case execution cost $T.e \leq T.p$. T may be invoked or *released* zero or more times, with any two consecutive invocations separated by at least $T.p$ time units. Each invocation of T is referred to as its *job*, is associated with a deadline that equals its period, and should complete executing within $T.p$ time units of its release. The ratio $T.e/T.p \leq 1.0$ denotes T's utilization $T.u$, which can be thought of as the *rate* at which T executes. Pfair algorithms are optimal for scheduling a task set τ of sporadic or rate-based tasks on M identical processors in that no job of any task would miss its deadline if the total utilization of τ is at most M.

Pfair algorithms achieve optimality by breaking each task into uniform-sized quanta and scheduling tasks one quantum at a time. Processor time is also allocated to tasks in discrete units of quanta. As such, all references to time will be non-negative integers. The time interval $[t, t+1)$, where t is a non-negative integer, is referred to as slot t. (Hence, time t refers to the beginning of slot t.) The interval $[t_1, t_2)$ consists of slots $t_1, t_1+1, \ldots, t_2-1$.

Each job of T consists of $T.e$ quanta. A task's quanta are numbered contiguously across its jobs (starting from one), thus, quanta $(j-1) \times T.e + 1$ through $j \times T.e$ comprise the j^{th} job T^j. The i^{th} quantum of T is denoted Q_T^i, and is associated with a release time and deadline, denoted $r(Q_T^i)$ and $d(Q_T^i)$, respectively, which define the window within which it should be scheduled. Release time and deadline for the i^{th} quantum that belongs to the j^{th} job of T (that is, $j = \lceil \frac{i}{T.e} \rceil$) are defined as follows.

$$r(Q_T^i) = r(T^j) - (j-1) \times T.p + \left\lfloor \frac{i-1}{T.u} \right\rfloor \wedge d(Q_T^i) = r(T^j) - (j-1) \times T.p + \left\lceil \frac{i}{T.u} \right\rceil \quad (1)$$

The windows of all the quanta that belong to the same job T^j are contained within the job's window, which spans $[r(T^j), d(T^j))$. Therefore, scheduling each quantum within its window (while ensuring that no two quanta of a task are scheduled concurrently) is sufficient to ensure that all job deadlines of all tasks are met. Windows of consecutive quanta are either disjoint or overlap in one time slot only. Refer to Fig. 2.

A *rate-based* task (also referred to as *intra-sporadic* task) generalizes a sporadic task by allowing quanta within a single job too to be delayed. Such delays are accommodated by shifting quantum windows to the right (by adding the delay to the formulas in (1)) as needed. Refer to the quantum windows of the third job in Fig. 2.

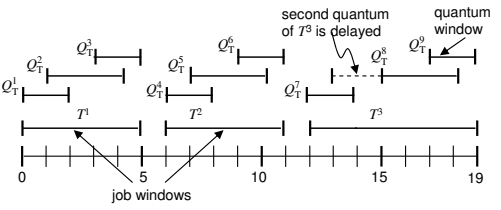

Fig. 2. First three jobs of a rate-based task T with $T.e = 3$ and $T.p = 5$. Job and quantum windows are depicted. Releases of the second and third jobs are delayed by one slot each. The second quantum in the third job is delayed by two slots.

On M processors, Pfair scheduling algorithms function by choosing at the beginning of each time slot, at most M eligible quanta of different tasks for execution in that time slot. Quanta are prioritized by their deadlines, with ties, if any, resolved using non-trivial tie-breaking rules. The most efficient of known Pfair algorithms has a per-slot time complexity of $O(M \log N)$, where N is the number of tasks [8]. Tasks may migrate under Pfair algorithms, and unless migration is allowed, optimality cannot be guaranteed.

3.2 Pfair Scheduling of DAG Tasks

A DAG task T can be Pfair scheduled by treating each of its subtasks as a stand-alone task of a sporadic task system and assigning the stand-alone task the subtask's execution cost and the DAG's period. Subtask dependencies imposed by the DAG can be taken care of by releasing the j^{th} instance T_i^j of subtask T_i only when both of the following hold: **(i)** the j^{th} instances of all the subtasks that T_i is dependent upon, as well as the previous instance, T_i^{j-1}, of T_i have completed execution, and **(ii)** at least $T.p$ time units have elapsed since the release time of the previous instance T_i^{j-1} of T_i. The optimality of Pfair scheduling immediately ensures that on $\lceil T.u \rceil$ processors, each instance of each subtask completes within $T.p$ time units of its release, providing the bound in the following theorem on the end-to-end latency through a DAG.

Theorem 1. *A DAG task T with total utilization $T.u \leq M$ can be Pfair scheduled on M identical processors such that its end-to-end latency is at most $S \times T.p$, where S is the number of subtasks in the longest, unweighted path in T.*

Since the end-to-end latency bound under Pfair depends only on the longest (unweighted) path to any leaf node, adding more nodes without increasing the path length, which amounts to increasing concurrency, would not worsen the latency bound. Hence, this approach can be said to *scale* with concurrency. On the other hand, every unit increase in the path length would, by the above theorem, worsen the latency bound by the DAG's period. This would certainly not be desirable if node execution costs are much less than the period. We address the issue of controlling the increase in latency with increasing path length in the next section. Our approach is based on grouping consecutive nodes on a sub-path into a single schedulable entity. Towards that end, we first consider scheduling linear chains of subtasks.

3.3 Scheduling Task Chains

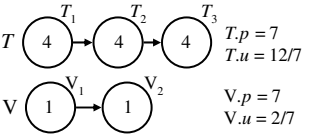

Fig. 3. Sample task chains

If the subtasks of a linear chain are Pfair scheduled as described in Sec. 3.2, then by Thm. 1, end-to-end latency for the chain could be up to the number of subtasks times the chain's period, which could be high in comparison to the chain's total execution cost. E.g., the subtasks in the chains in Fig. 3 could be Pfair-scheduled on two processors. However, the end-to-end chain latencies could be up to 21 and 14, respectively, which are a little too high.

It is quite easy to see that the two subtasks of V can be combined into a single sporadic task with an execution cost of two and a period of seven that replaces the subtasks and is scheduled as a single entity by a first-level scheduler. Allocations to V can internally be passed on to its two subtasks. Since V is guaranteed to receive two quanta within seven slots of each release, its end-to-end latency bound can be halved to seven.

The total utilization of T is $\frac{12}{7}$, which exceeds 1.0; hence, it is not clear whether the above straightforward approach of grouping a chain's subtasks applied to V extends to T. Scheduling chains with utilization greater than 1.0 is considered next.

Scheduling a task chain with utilization exceeding 1.0. We propose a two-level hierarchical approach for scheduling such task chains. Let S be a task chain with $S.u > 1.0$. Let $S.u^I = \lfloor S.u \rfloor$ and $S.u^f = S.u - S.I < 1.0$ denote the integral and fractional parts of $S.u$, respectively. (For T above, $T.u^I = 1$ and $T.u^f = \frac{5}{7}$.) In our approach, S is assigned $S.u^I + 1$ "fictitious" rate-based tasks (FTs) or virtual processors, $S.u^I$ of which are of unit capacity, and the final one, which is a fractional fictitious task ($\mathrm{FT}^{\mathrm{fr}}$, denoted $S.F$), has a utilization $1 \geq S.F.u \geq S.u^f$. FTs of S shall be scheduled along with stand-alone and other FTs by a first-level Pfair scheduler. A second-level scheduler local to S schedules the subtasks of S upon the time allocated to its FTs, while also controlling when the jobs and quanta of $S.F$ are released. For instance, $S.F$'s job may have to be postponed when the chain's release is postponed. Local scheduling within S shall be *preemptive* and prioritize any *ready* subtask that corresponds to an earlier instance over another that corresponds to a later instance. Then, one might expect that end-to-end latency for S should be bounded if $S.F.u = S.u^f$. However, it turns out that, for latency to be bounded, a slightly higher capacity, in the form of utilization greater than $S.u^f$ for their $\mathrm{FT}^{\mathrm{fr}}$ $S.F$, might be required for some chains S. In what follows, we provide an example of a chain for which no extra capacity is needed and a counterexample for which the minimal capacity does not suffice. Let $T_{i,j}^k$ denote the j^{th} quantum of the k^{th} instance of subtask T_i.

Example. Let T of Fig. 3 be allocated two FTs of utilizations 1.0 and $\frac{5}{7}$, respectively. Allocations due to the first FT are guaranteed every slot, whereas those due to the second whenever that task is scheduled by the first-level scheduler. It can be verified that if the task chain's releases are separated by *exactly* seven slots, then a pattern in which each instance receives five quanta in the first seven slots of its release and thereafter executes continuously until completion, with no allocation going unused, emerges. By this approach, the end-to-end latency for T is lowered to 14 from 21.

Counterexample. This seemingly effective approach can fail if allocations due to FT^{fr} cannot be put to use due to SCC. Consider Y with $Y.u = \frac{4}{3}$ in Fig. 4. The first schedule in the figure is when the capacity allocated to Y is $4/3$; hence, $Y.F.u = 1/3$. Each quantum of $Y.F$ can receive its allocation in any one of the three slots that span its window. If the allocations are always such that they cannot be put to use, as it is the case in the schedule depicted (solid lines in the second row), then latency for Y would grow without bound. In the example, $Y^5_{1,1}$ cannot be scheduled at time 13 because the previous instance of Y_1, Y^4_1, is executing at that time. It can be verified that delaying $Y.F$'s quanta do not prove to be effective either.

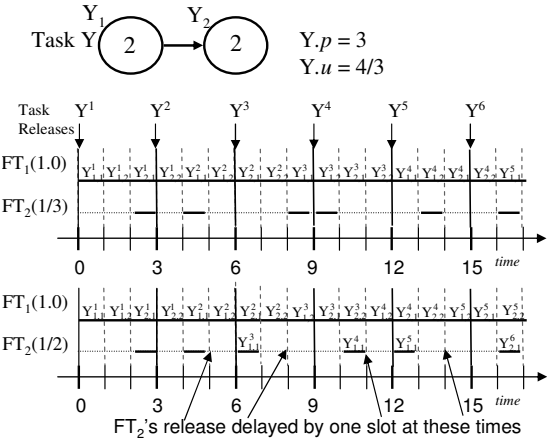

Fig. 4. Initial segments of schedules for the task chain Y when allocated processing capacities of $1\frac{1}{3}$ and $1\frac{1}{2}$

Bounding latency. One way of ensuring bounded latency for a task as above is to inflate the capacity allocated to it. The increase depends on the task parameters, and in most cases, is much less than rounding to the next integer. For Y above, inflating by $\frac{1}{6}$ from $1\frac{1}{3}$ to $1\frac{1}{2}$ suffices. A schedule for Y with the inflated allocation is also shown in Fig. 4. The extra capacity, if any, needed by a chain is determined in Thm. 2.

To see how inflating helps, note that in our example, when the fractional capacity is $\frac{1}{3}$, it is not possible to guarantee that allocations due to the FT^{fr} do not occur when a prior instance of the first subtask in the chain executes. For instance, if the third quantum due to $Y.F$ were available at time 6 or 7 (instead of 8) in the first schedule, then $Y^3_{1,1}$ could have executed concurrently with $Y_{2,2}$. Increasing the capacity to $\frac{1}{2}$ gives us the flexibility to postpone the release of the fractional task when wastage is possible, so that at least one quantum that can be put to use gets allocated within three slots of a release of Y.

Before we proceed further, some notation is in order. For any task chain C, let $C.e^f \stackrel{\text{def}}{=} C.e\%C.p$, and $C.e_i \stackrel{\text{def}}{=} \sum_{k=1}^{i} C_k.e$, the cumulative execution cost of its first i subtasks. Take $C.e_0 = 0$. Let $C.\sigma$ denote that subtask of C that would have commenced execution but not completed if C were allocated $C.e^f$ time slots. Formally, $C.\sigma \stackrel{\text{def}}{=} C_k$, where $k = (i | C.e_{i-1} < C.e^f \wedge C.e_i > C.e^f)$, if an i as defined exists, and $\emptyset$, otherwise. Finally, let $C.\sigma.e^1$ denote the number of quanta of $C.\sigma$ that are contained in the initial $C.e^f$ quanta of C, and $C.\sigma.e^2$, the remaining number of quanta of the same subtask. (If $C.\sigma = \emptyset$, then $C.\sigma.e^1 = C.\sigma.e^2 = 0$.) For T in Fig. 3, $T.e^f = 5$, $T.\sigma = T_2$, $T.\sigma.e^1 = 1$ and $T.\sigma.e^2 = 3$.

Since subtasks are prioritized by the instances they are part of, of all the pending instances, the latest instance of C, C^ℓ, is bound to receive the least allocation in the $C.p$ slots of its release. Our goal is to ensure that this allocation to C^ℓ is at least $C.e\%C.p$ quanta, by ensuring that at least this many quanta due to $C.F$ remain "usable."

If an instance of C, C^j, receives $C.e^f$ quanta in the first $C.p$ time slots of its release, then $C.\sigma^j$ would receive $C.\sigma.e^1$ of those, and the amount of execution still pending for $C.\sigma^j$ would be $C.\sigma.e^2$. If C^j can execute continuously until completion after $C.p$ slots, then $C.\sigma^j$ would complete in as many slots as it has pending quanta. Hence, our goal would be accomplished if the allocations needed for the $C.\sigma.e^1$ initial quanta of the next instance of subtask $C.\sigma$, $C.\sigma^{j+1}$, are guaranteed to be available in $C.p - C.\sigma.e^2$ slots following the completion of $C.\sigma^j$. As is formally proved below, this is possible if the processing capacity allocated to C is increased to the maximum of $C.u^f$ and $C.u^{f'} \stackrel{\text{def}}{=} \frac{C.\sigma.e^1}{C.p - C.\sigma.e^2}$.

Theorem 2. *A task chain C with $C.u > 1.0$ can be guaranteed an end-to-end latency bound of $\lceil C.u \rceil \times C.p$ if C is allocated a processing capacity due to at least $\lfloor C.u \rfloor$ unit capacity* FT*s and one* FT^{fr}*, $C.F$, with utilization $C.F.u = \max(\frac{C.e^f}{C.p}, \frac{C.\sigma.e^1}{C.p - C.\sigma.e^2})$.*

Proof. Let local scheduling within C be as described earlier. Let $n = \lceil C.u \rceil - \lfloor C.u \rfloor$. We prove the theorem by claiming that if a capacity as specified is allocated, then every instance of C executes **(a)** for at least $C.e^f \leq C.p$ slots in the first $n \times C.p$ slots of its release and **(b)** continuously thereafter until completion. The bound on latency would immediately follow. (Part (b) of the claim assumes that each instance of each subtask executes for exactly its execution cost, and each instance of C executes for exactly $C.e^f$ slots in the first $C.p$ slots. This assumption can be relaxed at the cost of a longer proof.)

The proof of the above claim is by induction on C's instances.

Base Case. Since the first instance of C is prioritized over the remaining instances, it can execute continuously from release until completion. Hence, it would receive all of the initial $C.p$ slots, which forms the base case.

Induction Step. Assume that the claim holds for the first $k-1$ instances (induction hypothesis, IH). We first prove Part (a), *i.e.*, show that the k^{th} instance executes for at least $C.e^f$ slots within $n \times C.p$ slots of its release. If $C.u$ is integral, then this part is vacuously true. Hence, assume otherwise, so $n = 1$. Let t denote the release time of C^k, $r(C^k)$. Because C's releases are at least $C.p$ slots apart, by IH, no subtask preceding $C.\sigma$ would be pending in any prior instance, and the prior instance of $C.\sigma$, $C.\sigma^{k-1}$, completes execution by time $t + C.\sigma.e^2$. Therefore, it suffices to ensure that the k^{th} instance of $C.\sigma$, $C.\sigma^k$, can receive at least $C.\sigma.e^1$ quanta in $[t+C.\sigma.e^2, t+C.p)$, while the k^{th} instances of the subtasks preceding $C.\sigma$ receive at least $C.e^f - C.\sigma.e^1$ quanta before $C.\sigma^k$. We consider two cases.

Case 1. $\frac{C.e^f}{C.p} \geq \frac{C.\sigma.e^1}{C.p - C.\sigma.e^2}$**.** In this case, $C.F.u = \frac{C.e^f}{C.p}$ and $C.F.p = C.p$. Therefore, since C's arrivals are separated by at least $C.p$ slots, $C.F$'s releases can be made to coincide with those of C. Hence, a new instance of $C.F$ is released at t. The number of quanta windows of $C.F$ fully contained in the interval $[t, t + C.p)$ is exactly $C.e^f$, its execution cost. Of these, at most $\lceil C.F.u \times C.\sigma.e^2 \rceil$ are contained (either fully or partially) in $[t, t + C.\sigma.e^2)$ in which $C.\sigma^{k-1}$ may execute. Therefore, the number of windows fully contained in $[t + C.\sigma.e^2, t + C.p)$ is at least $C.e^f - \lceil C.F.u \times C.\sigma.e^2 \rceil$. Since $C.\sigma.e^2 = C.\sigma.e - C.\sigma.e^1$ and $C.F.u = \frac{C.e^f}{C.p}$, we have

$$\#\text{ of quantum windows fully contained in } [t + C.\sigma.e^2, t + C.p) \geq C.e^f - \lceil \tfrac{C.e^f \times (C.\sigma.e - C.\sigma.e^1)}{C.p} \rceil. \quad (2)$$

By $\frac{C.e^f}{C.p} \geq \frac{C.\sigma.e^1}{C.p-C.\sigma.e^2}$, we have $\frac{C.\sigma.e-C.\sigma.e^1}{C.p} \times C.e^f \leq C.e^f - C.\sigma.e^1$ (cross-multiply and use $C.\sigma.e^2 = C.\sigma.e - C.\sigma.e^1$), substituting which in (2), we have the number of windows fully contained in $[t + C.\sigma.e^2, t + C.p)$ to be at least $C.\sigma.e^1$. Therefore, $C.\sigma^k$ can execute for $C.\sigma.e^1$ slots without conflict with $C.\sigma^{k-1}$, while the previous allocations due to $C.F$ can be made use of by the preceding subtasks to execute for $C.e^f - C.\sigma.e^1$ quanta.

Case 2. $\boldsymbol{\frac{C.e^f}{C.p} < \frac{C.\sigma.e^1}{C.p-C.\sigma.e^2}}$**.** In this case, cross-multiplying and rearranging terms in its condition, we have $\frac{C.e^f}{C.p} > \frac{C.e^f-C.\sigma.e^1}{C.\sigma.e^2}$. Hence, $C.F$'s capacity in this case, $\frac{C.\sigma.e^1}{C.p-C.\sigma.e^2}$, is greater than $\frac{C.e^f-C.\sigma.e^1}{C.\sigma.e^2}$. Also, $C.p = C.\sigma.e^2 + (C.p - C.\sigma.e^2)$. Therefore, to ensure that $C.\sigma^k$ does not become ready until its previous instance $C.\sigma^{k-1}$ completes at $t + C.\sigma.e^2$, $C.F$ can be throttled to receive allocations at a lower rate of $\frac{C.e^f-C.\sigma.e^1}{C.\sigma.e^2}$ in the first $C.\sigma.e^2$ slots of C^k's release, by releasing its job at t with execution cost $C.e^f - C.\sigma.e^1$ and deadline $t + C.\sigma.e^2$. This would result in an allocation of exactly $C.e^f - C.\sigma.e^1$ quanta in $[t, t + C.\sigma.e^2)$ which can be used by subtasks preceding $C.\sigma^k$. At time $t + C.\sigma.e^2$, when $C.\sigma^{k-1}$ completes execution, $C.F$'s job released at t would also complete. Hence, $C.F$'s capacity can be restored and a new job released for it at that time with execution cost $C.\sigma.e^1$ and deadline $t + C.p - C.\sigma.e^2$, which would lead to an allocation of $C.\sigma.e^1$ quanta for $C.\sigma^k$. Changing a task's utilization at job boundaries, as long as the total utilization of the task system does not exceed the processing capacity available, does not lead to any deadline misses under Pfair scheduling, as proved in [9]. Thus, $C.F$ can be made to receive exactly $C.e^f$ quanta and C^k to execute for $C.e^f$ slots in the first $C.p$ slots.

We are left with proving Part **(b)**. Suppose to the contrary that C^k does not execute continuously from time $t + n \times C.p$. Let t' denote the earliest time at or after $t + n \times C.p$ that C^k does not execute. Then, at t', either all the processors are busy executing prior instances of C, or some subtask of C^k has its previous instance still executing and hence cannot commence. The former implies that at least $\lfloor C.u \rfloor$ prior instances of C are still executing at t'. The release time of $(k - \lfloor C.u \rfloor)^{\text{th}}$ instance is at or before $t - C.p \times \lfloor C.u \rfloor$, and hence, by IH, completes at or before $t - C.p \times \lfloor C.u \rfloor + \lceil C.u \rceil \times C.p \leq t + C.p \leq t'$, a contradiction. Therefore, C^k's execution cannot be stalled due to lack of processors. We next show that C^k's execution is not blocked due to SCC either. Suppose not; let C_ℓ^k be the first subtask in C^k that is blocked because C_ℓ^{k-1} is still executing. Since $r(C^{k-1}) \leq t - C.p$, C_ℓ^{k-1} completes at $t - C.p + (C.e_\ell - C.e^f) + C.p = t + C.e_\ell - C.e^f$ (by IH). The completion time of $C_{\ell-1}^k$ is given by $t + (C.e_{\ell-1} - C.e^f) + C.p$, which, because $C_\ell.e \leq C.p$, is at least $t + (C.e_{\ell-1} - C.e^f) + C_\ell.e$, that is, at least $t + C.e_\ell - C.e^f$, the completion time of C_ℓ^{k-1}. Thus, C_ℓ^{k-1} completes before $C_{\ell-1}^k$, a contradiction again. The theorem follows. ■

Corollary 1. *The end-to-end latency of a task chain when it is Pfair-scheduled as a single entity is at most the latency when its subtasks are Pfair-scheduled independently.*

Proof. Follows from Thms. 1 and 2 because $C.u$ is at most the number of C's subtasks. ■

4 Managing Latency in Arbitrary DAGs

In this section, we first present a heuristic for lowering end-to-end latency in arbitrary DAGs. The basic idea is to construct one or more (sub-)task chains, each of which consists of one or more consecutive subtasks in a sub-path of a DAG, and is scheduled as a single entity. By Cor. 1, latency through the grouped subtasks is at most that when they are scheduled independently, and is likely to be much lower when grouped well. After dealing with task chain construction, we consider computing a bound on the end-to-end latency.

4.1 Chain Construction Heuristic

The basic idea is to iteratively do *depth-first-search* from candidate nodes (nodes with no incoming edges from subtasks not yet grouped) to identify the next best (sub-)path whose nodes can be grouped. Candidate paths are built such that no node in the path has an incoming edge from a node that is *not* in either the path under construction or a chain constructed in a previous step. This way, the chain sequence would be acyclic, which simplifies latency computation. Some criteria for identifying the best path are inflation to utilization, effectiveness in lowering latency, path length *etc*.

An example is provided in Fig. 5. Grouping starts from T_1. Two candidate paths from T_1 are $C_1 = T_1T_2T_4$ and $C_1' = T_1T_3$. These paths cannot be extended further since there are incoming edges to T_6 and T_5 from T_5 and T_2, respectively, that are not on their respective paths. $C_1.e^f = 5$, $C_1.u^f = \frac{5}{10}$, and $C_1.u^{f'} = \frac{2}{6}$. $C_1.u^f > C_1.u^{f'}$, so by Thm. 2, C_1's utilization need not be inflated. Since $C_1'.u = \frac{7}{10} < 1.0$, no inflation is needed for C_1', either. Of the two, we choose the longer path, C_1. The next candidate node is T_3, the path $T_3T_5T_6$ from which covers all the remaining nodes. This path too does not need inflation to utilization. It should be noted that if extending a path by one or more nodes leads to higher inflation to utilization, then our heuristic does not perform the extension.

One complication that arises when scheduling a set of chains constructed from a DAG is that any node in a chain (not necessarily the first) may have an incoming edge from any node (not necessarily the last) in a previous chain. Edges (T_1, T_3) and (T_2, T_5) are examples. Consequently, an instance of a chain may block after it commences. E.g., it can be verified that T_3^1 (in C_2) becomes ready at most six time units after T^1's release, and can complete by time 11, whereas T_5^1 (in C_2 again) may not be ready until time 14. If the blocking subtask is served by the FT^{fr}, then during the stall, the pending quanta of the FT^{fr} should be delayed (recall that a rate-based task's quanta can be delayed) to ensure that the allocations due to it do not go waste and latency is bounded. In our example, the release of the 5^{th} quantum of $C_2.F^1$ can be delayed until time 14.

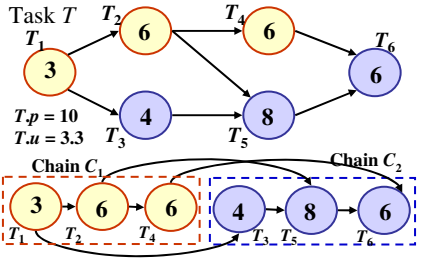

Fig. 5. Grouping subtasks into chains

```
procedure CONSTRUCT-CHAINS (DAG T)
1  while num_ungrouped_nodes > 0 do
2    for each candidate_node do
3      curr_chain := CHAIN-FROM
                     (candidate_node,
                     ∅);
4      if ISBETTER(curr_chain,
                   best_chain) then
5        update best_chain,
                 num_ungrouped_nodes,
                 list of grouped nodes;
       fi
     od
   od
6  coalesce as many adjacent chains as
   possible into a single one
```

```
procedure CHAIN-FROM(integer next_node, CHAIN
                     curr_chain) returns CHAIN
1  if there is an incoming edge from an ungrouped node to
                                   next_node then
2    return curr_chain;
   else
3    append next_node to curr_chain;
   fi
4  best_chain := curr_chain;
5  for each node ∈ adj_list(next_node) do
6    curr_chain := CHAIN-FROM( node, curr_chain);
7    if ISBETTER(curr_chain, best_chain) THEN
8      best_chain := curr_chain;
     fi
   od
9  return best_chain;
```

Fig. 6. Heuristic to group subtasks into chains

As can be seen from Sec. 4.2, latency through the DAG after grouping is at most 29 time units, which is only 1.26 times the length of the longest DAG path (23), while requiring no extra processing capacity. In comparison, if subtasks were statically mapped to processors, at least five processors ($>50\%$ extra capacity) would be required, and if subtasks were Pfair-scheduled without grouping, then the end-to-end latency could be up to 40.

Pseudo-code for the chain-construction heuristic is provided in Fig. 6 and should be self-explanatory. After basic chains are constructed, adjacent chains are coalesced if doing so has scope to lower either latency or inflation to utilization. When coalesced, nodes that belonged to different chains need not be on a path in the DAG. In such a case, an imposed dependency edge is introduced as appropriate. The complexity of the heuristic is $O(V(V + E))$, where V is the number of nodes, and E, the number of edges, in the DAG.

4.2 End-to-End Latency Bound

Our latency-computation algorithm for a DAG grouped into chains is based on that for determining the DAG's longest path. Pseudo-code is provided in Fig. 7. Nodes are considered in a topologically-sorted order by considering chains, and nodes within chains, in the order they were included while grouping. For each node, the latest time it would complete is determined by determining the completion time along each incoming edge.

Some aspects to note are as follows. First the time spent in a node can exceed its weight if the node is served by its chain's $\mathrm{FT}^{\mathrm{fr}}$. In such a case, the additional time taken can be computed from the release time and deadline (determined using Eq. (1)) of the quanta of the $\mathrm{FT}^{\mathrm{fr}}$ that will serve it. The accuracy of the end result can be increased if the latest completion time for the node is computed (as opposed to the time spent in it) by collectively considering all preceding nodes in the longest path to the node (through the incoming edge under consideration) that are in the same chain.

E.g., in Fig. 3, the latest completion time for T_2 would be ten if the sub-path consisting of its predecessor T_1 and itself is considered as a single unit. On the other hand, if the nodes are considered individually, and the maximum time through each is determined (using (1)), then the latest completion time for T_2 could be 11, which is a tad loose.

A second aspect is that a subtask earlier in a chain may be delayed by prior instances of later subtasks in the chain. Such delays should be properly accounted for. E.g., if T in Fig. 5 is released every ten slots, then T_3^3 cannot commence until time 29 (nine slots after T^3's release) even though it is ready at time 26, since T_6^1 and the first four quanta of T_5^2 may not complete until time 29. Hence, since T_3 is served by $C_2.F$ and $C_2.F.u = \frac{8}{10}$, and so can require up to five slots for its four quanta, the completion time of an instance of T_3 can be up to 14 slots from T's release time. If T_3 were to have an outgoing edge

```
procedure LATENCY-BOUND (DAG T, DAG_chains chains)
1  for each chain C ∈ chains do          ▷ taken in order
2    for each node of C do               ▷ taken in order
3      l_e_time[node] := T_node.e;
4      l_s_time[node] := 0;
5      for each incoming_edge into node do
         ▷ Compute the first and last quanta in the sub-path of
         ▷ all nodes (in C) in the longest path to node
         ▷ along incoming_edge
6        if incoming_edge is from pred_node in C then
7          st_q := first_q[pred_node];
8          end_q := st_q+path_e_in_chain[pred_node]
                    + T.e_node − 1;
         else
9          st_q := T.e_node − T_node.e+1;
10         end_q := T.e_node;
         fi
11       compute end_time and start_time along
                 incoming_edge using st_q, end_q, C.u^f,
                 C.u^f', C.F.u, l_s_time[pred_node], and Eq. (1);
12       if end_time > l_e_time[node] then
13         l_e_time[node] := end_time;
14         l_s_time[node] := start_time;
15         first_q[node] := st_q;
16         path_e_in_chain[node] := end_q − st_q+ 1;
         fi
       od
17     if l_e_time[node] > max_t then max_t := l_e_time[node]; fi
     od
     ▷ Adjust end_time to take into account delays due to
     ▷ earlier instances of later subtasks
18   for each node of C do               ▷ taken in reverse order
19     if l_s_time[node] > l_e_time[pred_node_in_chain] then
20       update l_e_time[pred_node_in_chain] and
         l_s_time[pred_node_in_chain] appropriately;
       fi
     od
   od
21 return max_t;
```

Fig. 7. Algorithm to compute end-to-end latency bound through a DAG decomposed into chains

to a later chain, then the ready time of the subtask that the edge is incident to should be taken as 14 and not 11. In Fig. 7, such delays are accounted for by the **for** loop beginning at line 20. It can be shown (by induction over chains) that latency is bounded despite such delays. A formal proof is omitted due to lack of space. The complexity of the algorithm is $O(E + V)$.

5 Empirical Evaluation

Experiments were conducted using randomly-generated task graphs to evaluate the efficacy of Pfair scheduling (with and without grouping) in lowering processor needs and latency.

Random DAGs were generated using the following configuration parameters: total number of DAG nodes (`total_nodes`), the upper limit on spatial concurrency (`max_conc`), and the maximum length (in nodes) of a concurrent path (`conc_len`). Fig. 8 provides an illustration. Nodes were added in the order of stages until the limit of `total_nodes` was reached. The number of nodes in a sub-stage of a concurrent stage was distributed uniformly between $0.6\times$`max_conc` and `max_conc`. When a new node was added, needed edges connecting it to those already present were inserted. All leaf nodes that existed after the addition of `total_nodes`-1 were linked to the final node. Cross edges connecting concurrent paths were randomly inserted with a uniform probability of 0.05. Each DAG's period was set to 20 and subtask execution costs were uniformly distributed between 6 and 15.

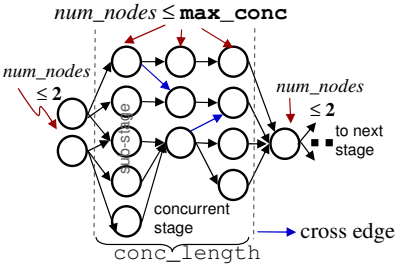

Fig. 8. Random DAG structure

For each DAG generated, we determined upper bounds on latencies under Pfair scheduling with and without grouping. We also determined the number of processors that would be needed if migration were disallowed. Results are shown in Fig. 9. Each value reported is the average determined for 10,000 DAGs. Inset (a) shows latency results for DAGs with 31 nodes (resp., 16 nodes) for `max_conc` values of 2,4, and 8, (resp., 2, 4, and 6) with `conc_len` set to three (resp., two) in each case. Our comparison metric is the ratio of latency under the considered approach to the longest weighted path through the DAG (a lower bound on latency). As expected, latencies are considerably and consistently lower when subtasks are grouped. Also as expected, when the number of nodes is constant, the efficacy of grouping decreases with increasing concurrency, though it is still better by over 30% even in the worst case reported. Turning to the processing needs of the various approaches, a processing capacity that equals the total utilization (whose average value is 17.5 and 8.5 for 31 and 16 nodes, respectively) suffices when nodes are ungrouped. When grouped, the average total inflation to a DAG is around 1% of the total utilization, while it is close to 50% under static mapping. Inset (b) shows the results for constant `max_conc` but varying `total_nodes`, and hence, varying path lengths. Results for other values of configuration parameters were similar, but could differ if subtask execution costs are chosen differently.

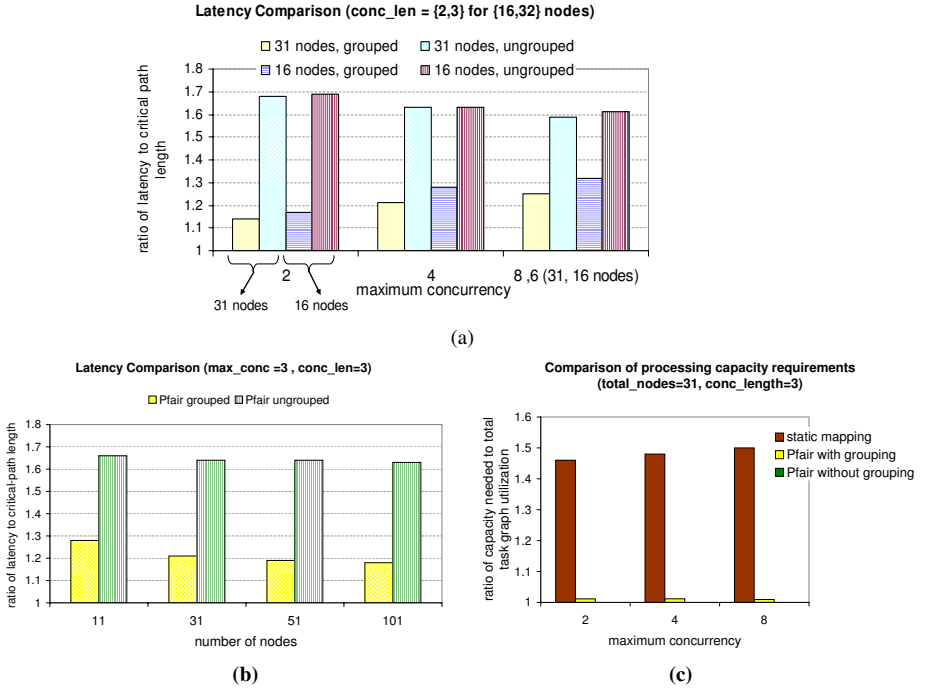

Fig. 9. Comparison of latencies and processor needs under different approaches

6 Related Work

Scheduling a task with concurrency modeled as a DAG on multiprocessors is well-studied, with a large body of work focused on minimizing the makespan of a single task instance. Since the problem is NP-complete for the general version and for most except a few simple variants [2], the focus has been on devising efficient heuristics. Refer to [7] for a survey.

Some later and recent work in this area has turned to scheduling recurring instances of DAG tasks. Most of the work is for DMMs and considers statically mapping tasks to processors under differing assumptions on task concurrency and structure. Scheduling DSP applications to maximize throughput is considered in [5]. The target platform in this work is closely-coupled, but has local processor memories and a segmented bus that make memory access times non-uniform. Optimizing latency under throughput constraint and vice versa by assigning parallelizable subtasks (which thereby allow data parallelism) to multiple processors is considered in [1]. Mapping a chain of data-parallel tasks to processors, including replicating subtasks for optimizing throughput, is the subject of [10], while evaluating latency-throughput tradeoffs that of [11].

There has also been work targeting specific architectures. E.g., scheduling on processors connected by point-to-point networks to meet the throughput requirement is considered in [4], while network of workstations are targeted in [12] for scheduling

video-processing algorithms to optimize the number of processors needed for a desired throughput and vice versa. Heuristics for mapping tasks of streaming applications for execution on workstations of a cluster is considered in [3].

All of the work referred to above is for DMMs in which inter-processor communication and migration overheads can be significant. Hence, applying techniques proposed therein for SMPs can be overkill.

7 Conclusion

We have proposed allowing the subtasks of a DAG task to migrate across the processors or cores of an SMP to enable meeting throughput requirements with fewer processors. We have also proposed a heuristic for lowering the end-to-end latency of a DAG and an algorithm for determining an upper bound on that measure under the scheduling approach proposed. Empirical evaluation using synthetic task graphs shows that our approaches can significantly lower processor needs, while incurring only a modest increase in latency in comparison to those that prohibit migration.

Some avenues for future work are as follows. First, the proposed algorithms can be extended to architectures in which not all but only subsets of cores share common caches and evaluated on a multicore test-bed. Second, latency computation can be incorporated within the grouping heuristic to construct better groups that can lower latency, and guarantees on performance that can be made in general can be determined. Finally, the latency computation algorithm can be extended for bursty arrivals and stochastic workloads.

References

1. Choudhary, A., Narahari, B., Nicol, D., Simha, R.: Optimal processor assignment for a class of pipelined computations. IEEE Transactions on Parallel and Distributed Systems 5(4), 439–445 (1994)
2. Garey, M., Johnson, D.: Computers and Intractability: a Guide to the Theory of NP-Completeness. W.H. Freeman, New York (1979)
3. Guirado, F., Ripoll, A., Roig, C., Luque, E.: Optimizing latency under throughput requirements for streaming applications on cluster execution. In: Proceedings of the IEEE International Conference on Cluster Computing, September 2005, pp. 1–10 (2005)
4. Hary, S., Ozguner, F.: Precedence-constrained task allocation onto point-to-point networks for pipelined execution. IEEE Transactions on Parallel and Distributed Systems 10(8), 838–851 (1999)
5. Hoang, P., Rabaey, J.: Scheduling dsp programs onto multiprocessors for maximum throughput. IEEE Transactions on Signal Processing 41(6), 2225–2235 (1993)
6. Kazempour, V., Fedorova, A., Alagheband, P.: Performance implications of cache affinity on multicore processors. In: Proceedings of the 14th International Conference on Parallel Computing, August 2008, pp. 151–162 (2008)
7. Kwok, Y.K., Ahmad, I.: Static scheduling algorithms for allocating derected task graphs to multiprocessors. ACM Computing Surveys 31(4), 406–471 (1999)
8. Srinivasan, A., Anderson, J.: Optimal rate-based scheduling on multiprocessors. In: Proceedings of the 34th ACM Symposium on Theory of Computing, May 2002, pp. 189–198 (2002)

9. Srinivasan, A., Anderson, J.: Fair scheduling of dynamic task systems on multiprocessors. Journal of Systems and Software 77(1), 67–80 (2005)
10. Subhlok, J., Vondran, G.: Optimal mapping of sequences of data parallel tasks. In: Proceedings of the 5th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, July 1995, pp. 134–143 (1995)
11. Subhlok, J., Vondran, G.: Optimal latency-throughput tradeoffs for data parallel machines. In: Proceedings of the 8th ACM Symposium on Parallel Algorithms and Architectures, June 1996, pp. 62–71 (1996)
12. Yang, M.-T., Kasturi, R., Sivasubramaniam, A.: A pipeline-based approach for scheduling video processing algorithms on NOW. IEEE Transactions on Parallel and Distributed Systems 14(2), 119–130 (2003)